Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells

Ran Mo*, Yuanfang Cai*, Rick Kazman†, Lu Xiao*

* Drexel University
Philadelphia, PA, USA
{rm859,yc349,lx52}@drexel.edu

†University of Hawaii & SEI/CMU
Honolulu, HI, USA
kazman@hawaii.edu

***Abstract*—In this paper, we propose and empirically validate a suite of *hotspot patterns*: recurring architecture problems that occur in most complex systems and incur high maintenance costs. In particular, we introduce two novel hotspot patterns, *Unstable Interface* and *Implicit Cross-module Dependency*. These patterns are defined based on Baldwin and Clark's design rule theory, and detected by the combination of history and architecture information. Through our tool-supported evaluations, we show that these patterns not only identify the most error-prone and change-prone files, they also pinpoint specific architecture problems that may be the root causes of bugginess and change proneness. Significantly, we show that 1) these structure-history integrated patterns contribute more to error- and change-proneness than other hotspot patterns, and 2) the more hotspot patterns a file is involved in, the more error- and change-prone it is. Finally, we report on an industrial case study to demonstrate the practicality of these hotspot patterns. The architect and developers confirmed that our hotspot detector discovered the majority of the architecture problems causing maintenance pain, and they have started to improve the system's maintainability by refactoring and fixing the identified architecture issues.**

***Keywords*—Software Architecture, Software maintenance, Software Quality**

I. Introduction

Much research in the field of defect prediction [8], [22], [21] and localization [14], [12], [19] has been proposed to identify problems in source code using structural metrics, evolution history, etc. In our recent work [31], we first explored the relation between file error-proneness and software architecture. In that work we proposed a new architecture model called the *Design Rule Space* (DRSpace). A DRSpace models software architecture as a set of design rules and modules. By calculating the interaction between DRSpaces and the set of a project's error-prone files, we observed that the most error-prone files are typically highly architecturally connected, and can be captured by just a few DRSpaces. We also observed that each such error-prone DRSpace has multiple architectural problems, or issues.

After examining hundreds of error-prone DRSpaces over dozens open source and commercial projects, we have observed that there are just a few distinct *types* of architecture issues, and these occur over and over again. We saw these issues in almost every error-prone DRSpace, in both open source and commercial projects. Most of these issues, although they are associated with extremely high error-proneness and/or change-proneness, cannot be characterized by existing notions such as code smells [7], [10] or anti-patterns [17], and thus are not automatically detectable using existing tools. On the other hand, a large portion of the source code "issues" detected by existing industry standard tools, such as Sonar[1] or Understand[2], are *not* causing notable maintenance problems. This, then raises a serious question for a project manager or architect: how to determine which parts of the code base should be given higher priority for maintenance and refactoring? Which "complex" files need to be fixed, as they are incurring a huge maintenance penalty, and which ones can be left alone?

Based on Baldwin and Clark's design rule theory [1] and basic software design principles, we summarize these recurring architecture issues (that are frequently the root causes of high-maintenance costs) into five architecture *hotspot* patterns, namely: (1) *Unstable Interface*, (2) *Implicit Cross-module Dependency*, (3) *Unhealthy Inheritance Hierarchy*, (4) *Cross-Module Cycle*, (5) and *Cross-Package Cycle*. Of these five patterns, the first four are defined at the file level, and the last one is defined at the package level. (1) is defined based on the rationale that important interfaces (design rules) should be stable [1]. (2) is defined based on the concept of *true modules*, as described by design rule theory [1], [26], and revealed by a design rule hierarchy clustering [30]. This pattern aims to reveal hidden dependencies that connect modules that appear to be mutually independent [29]. (3) detects architecture structures that violate either design rule theory or Liskov Substitution principles[3]. (4) and (5) are based on well-known rationale of forming a proper hierarchy structure among modules and packages.

We have formalized these five patterns because we regularly encounter them in numerous error-prone spaces. Most of these patterns, such as Unstable Interface, Implicit Cross-module Dependency, and Unhealthy Inheritance Hierarchy, have not been formally defined before and not detectable by existing tools. For Implicit Cross-module Dependency and Cross-module Dependencies, we define "modules" as mutually independent file groups revealed by a design rule hierarchy (DRH) [30]. Like any catalog of patterns, we can not claim completeness, just observational adequacy. We are open to add more hotspot patterns if more recurring and high-maintenance architecture problems are observed in the future. Furthermore, based on our formalization, we are able to easily add new hotspot patterns to our tool, and to automatically detect these

[1] http://www.sonarqube.org/
[2] https://scitools.com/
[3] https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start

issues in any code base.

Using nine open source projects and one commercial project, we evaluated these patterns both quantitatively and qualitatively. In our quantitative analysis, we investigate: 1) if hotspots really capture the architecture problems that incur expensive maintenance costs in terms of error-proneness and change-proneness; 2) if the more patterns a file is involved in, the more error-prone and/or change-prone it is; (3) if different types of hotspots have different influence on file's overall error-proneness and change-proneness.

Our result shows that compared with average files in a project, the files involved in these patterns have significantly higher bug and change rates. By considering files involved in 0 to 4 file-level hotspot patterns, we observed that their bug rate and change rate increase dramatically as the number of hotspot patterns they are involved in increases. And of all the 4 file-level hotspot patterns, Unstable Interface turns out to have the most significant contribution to error- and change-proneness.

For our qualitative evaluation, we conducted an industrial study to determine the usefulness of these hotspot patterns in practice. We wanted to evaluate if these patterns reveal major architecture problems that are valuable to the architect and developers, and if, in addition to identifying *where* to refactor, these patterns can provide clues about *how* to refactor. In this case study, we identified two instances of Unstable Interface patterns and one instance of Implicit Cross-module Dependency, influencing more than 100 files. The architect and developers in our study confirmed that our hotspot detector discovered a majority of the architecture problems that are causing maintenance pain. Furthermore based on the nature of the patterns discovered, they were able to identify the hidden dependencies behind the Implicit Cross-module Dependency issue, and the two interfaces that have grown into "God" interfaces, thus needing to be refactored. They have started to improve the maintainability of their system by refactoring and fixing these architecture issues.

From the results of our evaluation, we have obtained positive answers to the our research questions. Hotspot patterns are strongly correlated with higher bug frequencies, change frequencies, and with the effort to fix bugs and make changes. And the more architectural issues a file is involved with, the worse it fares, in terms of bugs and effort. Furthermore, in our qualitative analysis we demonstrated that our approach is effective in helping real-world developers find the structural problems contributing to maintenance effort and helping them conduct their refactoring.

The rest of this paper is organized as follows: Section 2 presents several background concepts. Section 3 describes the theory and formalism underpinning our hotspot detection approach. Section 4 describes the details of our hotspot detection tool. Section 5 presents our evaluation methods, analysis, and results. Section 6 discusses the strengths and limitations of our tool. Section 7 discusses related work and Section 8 concludes.

II. Background

In this section, we introduce the fundamental concept behind hotspot patterns: *Design Rule Space* (DRSpace). In our recent work [31], we proposed DRSpaces as a new architecture model. Instead of viewing software architecture as simply a set of components and relations, we consider that architecture is structured by *design rules* and *independent modules*, following Baldwin and Clark's design rule theory [1].

Design rules reflect the most important architectural decision that decouple the rest of the system into *independent modules*. In a software system, a design rule is usually manifested as an interface or abstract class. For example, if an Observer Pattern [9] is applied, then there must exist an observer interface that decouples the subject and concrete observers into independent modules. As long as the interface is stable, addition, removal, or changes to concrete observers should not influence the subject. In this case, the observer interface is considered to be a *design rule*, decoupling the subject and concrete observers into two independent *modules*.

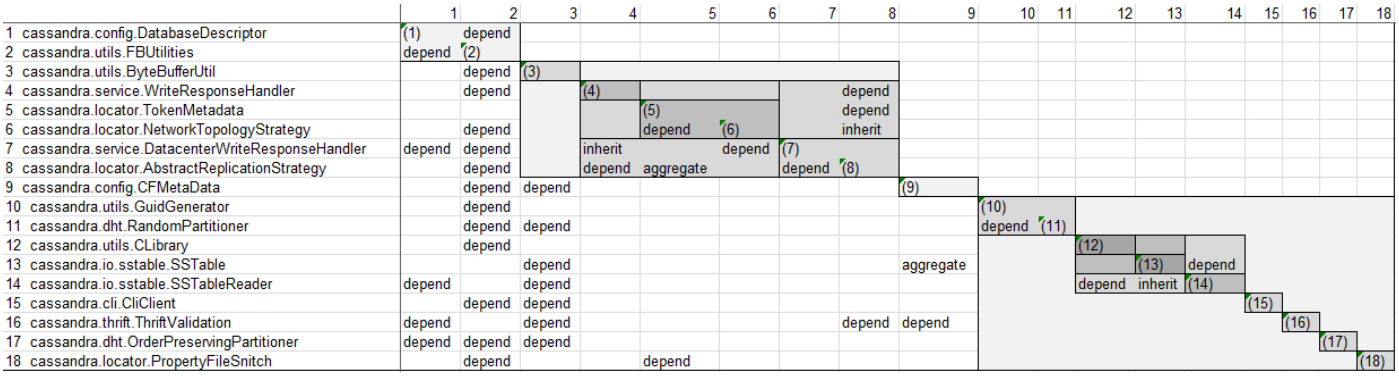
Consider another example: if a system applies a pipe-and-filter architecture pattern, then all the concrete filters become independent modules, connected by pipes. Reflected in code, the abstract Pipe class can be considered as an instance of design rule. Since a system can apply multiple design patterns, each pattern forms its own DRSpace. Accordingly, we proposed that a software architecture should be viewed as a set of overlapping DRSpaces.

A DRSpace contains a set of files and a selected set of relations, such as inheritance, aggregation, dependency. These files are clustered into a *design rule hierarchy* (DRH) [3], [30], [4] to manifest the existence of design rules and independent modules. The DRH algorithm clusters the files in a DRSpace into a special form of hierarchical structure with the following features: 1) the first layer of the architecture contains the most influential files in the system, such as important base classes, key interfaces, etc. These files are called the *leading files*. 2) Files in higher layers should not depend on files in lower layers. 3) Files within the same layer are grouped into mutually independent *modules*. If the system is designed with key architectural design rules, then the files reflecting these design rules will be among the leading classes.

We visualize a DRSpace using *Design Structure Matrix (DSM)*. A DSM is a square matrix, whose rows and columns are labeled with the files of the DRSpace in the same order. If a cell in row x, column y, c:(rx,cy), is marked, it means that file x is structurally related to file y or evolutionarily related (i.e., file x and file y have changed together in the evolutionary history). The cells along the diagonal model self-dependency. Using our tool, Titan [32], the user can view and manipulate DRSpaces.

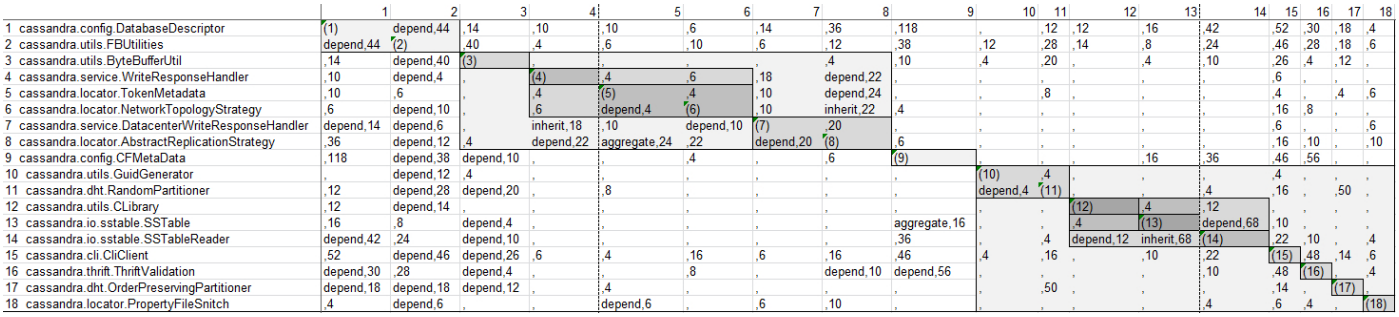
The DSM in Figure 1 presents a DRSpace clustered into a DRH with 4 layers: l1: (rc1-rc2), l2: (rc3-rc8), l3: (rc9), l4: (rc10-rc18). The first layer l1 contains the most influential design rules that should remain stable and they do not depend on lower layer files. Files in the second layer l2 only depend on first layer files. Similarly, files in the third layer l3 only depend on the first two layers. Taking the second layer l2 as an example, there are 2 mutually independent modules: m1: (rc3), and m2: (rc4-rc8). We can recursively apply DRH clustering on a complex module. For example, in m2, there are two inner layers: (rc4-rc6) and (rc7-rc8).

The cells in a DSM can be annotated to illuminate relations among the files. For example, cell(r3,c2) in Figure 1 is marked



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	cassandra.config.DatabaseDescriptor	(1)	depend																
2	cassandra.utils.FBUtilities	depend	(2)																
3	cassandra.utils.ByteBufferUtil		depend	(3)															
4	cassandra.service.WriteResponseHandler		depend		(4)				depend										
5	cassandra.locator.TokenMetadata					(5)			depend										
6	cassandra.locator.NetworkTopologyStrategy		depend			depend	(6)		inherit										
7	cassandra.service.DatacenterWriteResponseHandler	depend	depend		inherit		depend	(7)											
8	cassandra.locator.AbstractReplicationStrategy		depend		depend	aggregate		depend	(8)										
9	cassandra.config.CFMetaData		depend	depend						(9)									
10	cassandra.utils.GuidGenerator		depend								(10)								
11	cassandra.dht.RandomPartitioner		depend	depend							depend	(11)							
12	cassandra.utils.CLibrary		depend										(12)						
13	cassandra.io.sstable.SSTable			depend						aggregate				(13)	depend				
14	cassandra.io.sstable.SSTableReader	depend		depend									depend	inherit	(14)				
15	cassandra.cli.CliClient		depend	depend												(15)			
16	cassandra.thrift.ThriftValidation	depend		depend					depend	depend							(16)		
17	cassandra.dht.OrderPreservingPartitioner	depend	depend	depend														(17)	
18	cassandra.locator.PropertyFileSnitch		depend			depend													(18)

Fig. 1: DRSpace clustered into DRH Structure with only structure relations



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	cassandra.config.DatabaseDescriptor	(1)	depend,44	,14	,10	,10	,6	,14	,36	,118		,12	,12	,16	,42	,52	,30	,18	,4
2	cassandra.utils.FBUtilities	depend,44	(2)	,40	,4	,6	,10	,6	,12	,38	,12	,28	,14	,8	,24	,46	,28	,18	,6
3	cassandra.utils.ByteBufferUtil	,14	depend,40	(3)					,4	,10	,4	,20		,4	,10	,26	,4	,12	
4	cassandra.service.WriteResponseHandler	,10	depend,4		(4)	,4	,6	,18	depend,22							,6			
5	cassandra.locator.TokenMetadata	,10	,6		,4	(5)	,4	,10	depend,24			,8				,4		,4	,6
6	cassandra.locator.NetworkTopologyStrategy	,6	depend,10		,6	depend,4	(6)	,10	inherit,22	,4						,16	,8		
7	cassandra.service.DatacenterWriteResponseHandler	depend,14	depend,6		inherit,18	,10	depend,10	(7)	,20							,6			,6
8	cassandra.locator.AbstractReplicationStrategy	,36	depend,12	,4	depend,22	aggregate,24	,22	depend,20	(8)	,6						,16	,10		,10
9	cassandra.config.CFMetaData	,118	depend,38	depend,10			,4		,6	(9)				,16	,36	,46	,56		
10	cassandra.utils.GuidGenerator		depend,12	,4							(10)	,4				,4			
11	cassandra.dht.RandomPartitioner	,12	depend,28	depend,20		,8					depend,4	(11)			,4	,16		,50	
12	cassandra.utils.CLibrary	,12	depend,14										(12)	,4	,12				
13	cassandra.io.sstable.SSTable	,16	,8	depend,4						aggregate,16			,4	(13)	depend,68	,10			
14	cassandra.io.sstable.SSTableReader	depend,42	,24	depend,10						,36		,4	depend,12	inherit,68	(14)	,22	,10		,4
15	cassandra.cli.CliClient	,52	depend,46	depend,26	,6	,4	,16	,6	,16	,46	,4	,16		,10	,22	(15)	,48	,14	,6
16	cassandra.thrift.ThriftValidation	depend,30	,28	depend,4		,8			depend,10	depend,56					,10	,48	(16)		,4
17	cassandra.dht.OrderPreservingPartitioner	depend,18	depend,18	depend,12		,4						,50				,14		(17)	
18	cassandra.locator.PropertyFileSnitch	,4	depend,6			depend,6		,6	,10						,4	,6	,4		(18)

Fig. 2: DRSpace clusterd into DRH Structure with structure relations and evolutionary history

with "depend", which means *cassandra.utils.ByteBufferUtil* depends on *cassandra.utils.FBUtilities*. A DRSpace can also reveal the evolutionary coupling between files. In Figure 2, we see cells with a number representing the number of times that these two files changed together in the project's evolution history. The cell with just a number means there is no structural relation between these two files, but they have nonetheless changed together. For example, cell(r3,c1) is only marked with "14", which means there is no *structural* relation between *cassandra.utils.ByteBufferUtil* and *cassandra.config.DatabaseDescriptor*, but they have changed together 14 times, according to the revision history. A cell with a number and text means that the two files have both structural and evolutionary coupling. For example, cell(r9,c2) is marked with "depend,38", which means that *cassandra.config.CFMetaData* depends on *cassandra.utils.FBUtilities*, and they have changed together 38 times.

III. Architecture Issues

According to Baldwin and Clark's design rule theory [1], a well-modularized system should have the following features: first, the design rules have to be stable, that is, neither error-prone nor change-prone. Second, if two modules are truly independent, then they should only depend on design rules, but not on each other. More importantly, independent modules should be able to be changed, or even replaced, without influencing each other, as long as the design rules remain unchanged.

From the numerous DRSpaces we have processed, we have observed that files in DRSpaces where these principles are violated are associated with high error-proneness and change-proneness. We summarize the forms of violations that occur repetitively into a suite of *Hotspot Patterns*. Next we define some basic terms and introduce the rationale and formalization of these hotspot patterns.

A. Base Definitions

We use the following terms to model the key concepts of a DRSpace: F—the set of all the files in the DRSpace: $F = \{f_i \mid i \in \mathbb{N}\}$

M—the set of independent modules within layers: $M = \{m_i \mid i \in \mathbb{N}\}$

fm_i—a file in an independent module, m_i.

$cm(f)$—the most inner module that contains file f.

We use the following notions to model structural and evolutionary relations among files of a DRSpace:

$depend(x, y)$: x depends on or aggregates y

$inherit(x, y)$: x inherits from or realizes y

$nest(x, y)$: x is the inner class of y

$\#cochange(x, y)$: the number of times x was changed together with y in a given period of time.

$$SRelation(x, y) = depend(x, y) \vee depend(y, x) \vee inherit(x, y) \vee inherit(y, x) \vee nest(x, y) \vee nest(y, x)$$

B. Hotspot Patterns

For each hotspot pattern, we now introduce their rationale, description, and formalization.

1. Unstable Interface.
Rationale: Based on design rule theory, the most influential files in the system—usually the design rules—should remain stable. In a DRSpace DSM, the files in the first layer are always the most influential files in the system, that is, they are the leading files of the DRSpace. In most error-prone DRSpaces, we observe that these files frequently change with other files, and thus have large co-change numbers recorded in the DSM. Moreover, such leading files are often among the most error-prone files in the system, and most other files in their DRSpace are also highly error-prone. We formalize this phenomenon into an *Unstable Interface* pattern.

Description: If a highly influential file is changed frequently with other files in the revision history, then we call it an *Unstable Interface*.

Formalization:
$Impact_{thr}$: the threshold of the impact scope of a file, f_i. If the number of files that structurally depend on it is larger than the threshold, then we consider it as a candidate unstable interface. In practice this threshold is larger than 50% of the number of files in the DRSpace. And typically the files with largest impact scope are the leading files.

$cochange_{thr}$: the threshold for the frequency of two files changing together. If two files changed together more than the threshold, we say that they are evolutionarily coupled.

$Change_{thr}$: the threshold of the number of files which change together with a file, f_i, more than $cochange_{thr}$ times.

For the file f_1, if there are more than$Impact_{thr}$ files which structurally depend on it and there are more than $Change_{thr}$ files which changed together with it more than $cochange_{thr}$ times, we consider it as an *Unstable Interface*. Formally:

$\exists f_i, f_1 \in F \mid |SRelation(f_i, f_1)| > Impact_{thr} \wedge |\#cochange(f_i, f_1) > cochange_{thr}| > Change_{thr}$, where $i \in [2, 3, 4, ..., n]$, n is the number of files in DRSpace DSM. $Impact_{thr}$, $cochange_{thr}$ and $Change_{thr}$ are the thresholds that can be adjusted by an analyst.

In Figure 2, we can take file *FBUtilities* as an example, which is one of the leading files. In this case, let $Impact_{thr}$ be 10, $Change_{thr}$ be 8, and let $cochange_{thr}$ be 4. Thus *FBUtilities* becomes an *Unstable Interface*. Because there are more than 10 files in this DRSpace that structurally depend on it and there are more than 8 files which changed together with it more than 4 times in the project's revision history.

2. Implicit Cross-module Dependency.
Rationale: Truly independent modules should be able to change independently from each other. If two structurally independent modules in the DRSpace are shown to change together frequently in the revision history, it means that they are not truly independent from each other. We observe that in many of these cases, the modules have harmful *implicit* dependencies that should be removed [24].

Description: Consider a DRSpace where file groups within a layer form *true modules* that are mutually independent from each other. Suppose there are two independent modules within the same layer, m_1, m_2. If, for all the files in m_1, there is no structural relation with any of the files in m_2, but there exist files in m_1 which change together with one or more files in m_2 more than $cochange_{thr}$ times, then we consider the two modules follow a *Implicit Cross-module Dependency* pattern.

Formalization:
$\exists m_1, m_2 \in M \wedge \forall fm_i \in m_1 \wedge \forall fm_j \in m_2 \mid \neg SRelation(fm_i, fm_j) \wedge (\exists fm_1 \in m_1 \wedge \exists fm_2 \in m_2 \mid \#cochange(fm_1, fm_2) > cochange_{thr})$, where $i, j \in [1, 2, 3, ..., n]$, $cochange_{thr}$ is a threshold that can be adjusted by an analyst.

In the DRSpace of Figure 2, the bottom layer: (rc10-rc18) contains 6 mutually independent modules: m1:(rc10-rc11), m2:(rc12-rc14), m3:(rc15), m4:(rc16), m5:(rc17), m6:(rc18). All these modules should be able to change independently from each other, however, the cells annotated with a number (more than $cochange_{thr}$) crossing modules in this layer reveal that these modules do change together in the history, and that could be considered as a *Implicit Cross-module Dependency*. In this work we assume $cochange_{thr}$ to be 4.

3. Unhealthy Inheritance Hierarchy.
Rationale: We have encountered a surprisingly large number of cases where basic object-oriented design principles are violated in the implementation of an inheritance hierarchy. The two most frequent problems are: (1) a parent class depends on one of its children; (2) a client class of the hierarchy depends on both the base class and all its children. The first case violates the design rule theory since a base class is usually an instance of design rules and thus shouldn't depend on subordinating classes. Both cases violate Liskov Substitution principle since where the parent class is used won't be replaceable by its subclasses. In these cases, both the client and the hierarchy are usually very error-prone.

Description: We consider an inheritance hierarchy to be problematic if it falls into one of the two cases:

1) Given an inheritance hierarchy containing one parent file, f_{parent}, and one or more children, F_{child}, there exists a child file f_i satisfying $depend(f_{parent}, f_i)$
2) Given an inheritance hierarchy containing one parent file, f_{parent}, and one or more children, F_{child}, there exists a client f_j of the hierarchy, that depends on both the parent and all its children.

Formalization:
$\exists f_{parent}, F_{child} \in F \wedge \exists f_i \in F_{child} \mid depend(f_{parent}, f_i) \vee \exists f_j \in F \mid depend(f_j, f_{parent}) \wedge \forall f_i \in F_{child} \mid depends(f_j, f_i)$, where $i, j \in [1, 2, 3, ..., n]$, $f_j \notin F_{child}$ and $f_j \neq f_{parent}$.

In Figure 2, we displayed some *Unhealthy Inheritance Hierarchy*. For example, *SSTableReader* is the child file of *SSTable*, but its parent file depends on it. Therefore, we consider this as an *Unhealthy Inheritance Hierarchy* pattern.

4. Cross-Module Cycle.
Rationale: We have observed that not all cyclical dependencies among files are equally harmful, and that the cycles between modules—defined in the same way as in the Implicit Cross-module Dependency—in a DRSpace are associated with more

defects. We thus define Cross-Module Cycle as another hotspot pattern.

Description: If there is a dependency cycle and not all the files belong to the same closest module, we consider this to be a *C*ross-Module Cycle pattern.

Formalization:
$\exists f_i \in F \mid depend(f_1, f_2) \wedge depend(f_2, f_3) \wedge \cdots \wedge depend(f_{n-1}, f_n) \wedge depend(f_n, f_1) \wedge \neg(cm(f_1) = cm(f_2) = \cdots = cm(f_n))$, where $i \in [1, 2, 3, ..., n]$ and $n \geq 2$, n is the number of files in the cycle.

In Figure 2, we have displayed some *Cross-Module Cycles*. For example, $AbstractReplicationStrategy \rightarrow DatacenterWriteResponseHandler \rightarrow NetworkTopologyStrategy \rightarrow TokenMetadata \rightarrow AbstractReplicationStrategy$ form a Cross-Module Cycle.

5. Cross-Package Cycle.
Rationale: Usually the package structure of a software system should form a hierarchical structure. A cycle among packages is typically considered to be harmful.

Description: Given two packages P_1, P_2 of the DSM, there exists files f_1 in P_1 and f_2 in P_2. Now, given some files f_j in P_2 and f_i in P_1, if $depend(f_1, f_j)$ and $depend(f_2, f_i)$, then we can say that these two packages create a *Package Cycle*, that is, a cycle of dependencies between the *packages*.

Formalization:
$\exists f_1, f_i \in P_1 \wedge \exists f_2, f_j \in P_2 \mid depend(f_1, f_j) \wedge depend(f_2, f_i)$, where P_1, P_2 are the packages of the system, $i, j \in [1, 2, 3, 4, ..., n]$, n is the number of files in the system.

IV. Tool Support

We have created a tool, called the *Hotspot Detector* that automatically detects instances of these hotspot patterns. The tool takes the following three inputs:

1) A DSM file that contains the structural dependencies among the files, which we call a SDSM file. This DSM file can be generated using Titan [32]. Titan takes dependency information output by a reverse engineering tool, and translates it into a SDSM file.
2) A DSM file that contains the evolutionary coupling information of files within a DRSpace, which we call a HDSM file. This DSM is organized such that a number in a cell denotes the number of times two files have changed together. A HDSM can be generated by Titan from a revision history record, such as a SVN log.
3) A clustering file that contains the clustering information of the files. To detect a *Cross-Package Cycle*, a clustering file that contains the package structure of the files is used. To detect other types of hotspot patterns, we are using a clustering file that represents the DRH clustering.

Given these inputs, the Hotspot detector will output a summary of all the architecture issues, the files involved in each issue, and the DSMs containing the DRSpaces with these issues. These DSMs can be viewed using Titan's GUI.

V. Evaluation

In this section, we report our evaluation subjects, methods, quantitative and qualitative evaluation results.

A. Subjects

We choose nine Apache open source projects and one commercial project as our subjects. We selected these Apache projects to cover a wide variety of domains: Avro[4] is a serialization system; Cassandra[5] is a distributed database; Camel[6] is a integration framework based on known Enterprise Integration Patterns; CXF[7] is a services framework; Hadoop[8] is a tool for distributed processing of large data sets; HBase[9] is the Hadoop database; Ivy[10] is a dependency management tool for software projects. OpenJPA[11] is a Java persistence project. PDFBox[12] is a library for manipulating PDF documents. For each project we obtained a snapshot of their latest release. In Table I, we list some basic facts about these projects. In our evaluation, we used their revision histories, issue records and source code to obtain the HDSM, SDSM and clustering files. Given these inputs, our tool calculated all the architecture issues and involved files for each project snapshot.

TABLE I: Subject Projects

Subjects	#Files	#Commits	#BugIssues	SLOC
Avro-1.7.6	253	1488	629	180K
Camel-2.11.1	1203	17706	2110	869K
Cassandra-1.0.7	775	6738	1922	135k
CXF-2.7.10	3062	27266	3438	804K
Hadoop-2.2.0	1817	16003	4661	2.3M
HBase-0.94.16	1958	14861	4762	688K
Ivy-2.3.0	606	3799	799	308K
OpenJPA-2.2.2	1761	6744	1500	498k
PDFBox-1.8.4	589	1798	1098	116k
Commercial	797	2756	1079	56k

B. Evaluation Methods

For our quantitative analysis, we applied our hotspot detection tool to the nine Apache projects and one commercial project. For each of these projects, our tool detects all the files involved in each issue. For our qualitative analysis, we applied our architecture issue detection tool to the commercial project and sent the detected architecture issues back to the project's architect and developers. Subsequent to sending the results we conducted a brief survey and an interview with the architect.

To quantify error-proneness and change-proneness, we first define the following four measures: 1) bug frequency (bugFreq); 2) bug churn: the number of lines of code committed to fix a bug (bugChurn); 3) change frequency (changeFreq), and 4) change churn: the number of lines of code committed to make a change (changeChurn). These measures are calculated

[4]http://avro.apache.org/
[5]http://cassandra.apache.org/
[6]http://camel.apache.org/
[7]http://cxf.apache.org/
[8]http://hadoop.apache.org/
[9]http://hbase.apache.org/
[10]http://ant.apache.org/ivy/
[11]http://openjpa.apache.org/
[12]http://pdfbox.apache.org/

by mining change logs in the version control system and issue tracking system used by the project.

Table II depicts a brief summary of the detected architecture hotspots for each project. Some files are involved in multiple architecture hotspots, so the total '$\#Files$' means the total number of *distinct* files involved in all hotspots. The first observation to make is that architecture hotspots are indeed ubiquitous. Every project had at least one instance of every type of hotspot, and in some cases there were thousands of instances and thousands of files implicated. Architecture hotspots are real, and they are not rare.

C. Quantitative Analysis

We assess the following questions to evaluate our approach quantitatively:

RQ1. *DO the files involved in architecture issues incur more errors and cause significantly more maintenance effort than average files?* The answer to this question reveals if the architecture issues we detect are real issues with high maintenance costs.

RQ2. *If a file is involved in* greater numbers *of architecture issues, then is it more error-prone/change-prone than average files?* The answer to this requestion will indicate how these issues impact the error- and change- proneness of files.

RQ3. *Which types of architecture issues will cause more error-proneness and change-proneness in files?* We investigate this question to understand if some issues have more significant impact than others.

We defined a suite of measures and conducted statistical analysis to answer these questions, as we will explain. These questions are addressed by various measures of the outputs of the hotspot detection tool, and statistical analysis of the results.

Given our measures of maintenance effort—bugFreq (BF), bugChurn (BC), changeFreq (CF), changeChurn (CC)—we can now perform a number of quantitative analyses to understand the relationship between these measures and the detected occurrences of architecture issues in a project.

To answer the first research question, for each measure, we calculate the *increase* of its average value ($measure_i\_inc$). Let the measure's average value over all files be $avg\_measure_i$ and measure's average value for files involved in architecture issues be $avg\_arch\_measure_i$. Then we calculated:

$$measure_i\_inc = \frac{avg\_arch\_measure_i - avg\_measure_i}{avg\_measure_i} \times 100\%.$$

For each measure, Table III reports the increase in the value of the measure. Note that the measures all increased, over all projects, as expected. Specifically these results demonstrate that the average measure for files involved in architecture issues is larger than the average measure over all files. The smallest increase is 39.49%, which is the bugFreq (BF) of the OpenJPA project. The greatest increase is the changeFreq of HBase, which is 219.88%. Our results indicate that the files involved in architecture issues are more error-prone than average files. Hence these files will lead to more maintenance effort. To substantiate this claim we employ the *Paired t-Test* to test whether the population of $avg\_arch\_measure_i$ is significantly different than the population of $avg\_measure_i$ over the 10 projects.

Null Hypothesis: *H_0, the population of $avg\_measure_i$ is the same as the population of $avg\_arch\_measure_i$.*

Alternative Hypothesis: *H_1, population of $avg\_measure_i$ is different from the population of $avg\_arch\_measure_i$.*

The last row of Table III shows that, for all measures, H_1 is accepted. This means that there exists a significant difference between $avg\_measure_i$ and $avg\_arch\_measure_i$ across all 10 projects. The four measures are consistently and significantly greater for the files that exhibit architecture issues. Therefore, we have strong evidence to believe that files exhibiting architecture issues will cause more project maintenance effort.

TABLE III: Increase in the measures' average values for the 10 projects

Subjects	BF_inc	BC_inc	CF_inc	CC_inc
Avro-1.7.6	94.86%	80.04%	86.96%	78.41%
Camel-2.11.1	49.35%	48.31%	47.52%	44.11%
Cassandra-1.0.7	65.64%	64.80%	67.42%	66.12%
CXF-2.7.10	76.97%	70.88%	73.46%	67.98%
Hadoop-2.2.0	143.88%	136.50%	143.28%	126.76%
HBase-0.94.16	201.61%	219.88%	210.00%	210.04%
Ivy-2.3.0	87.23%	88.54%	76.87%	78.58%
OpenJPA-2.2.2	39.49%	68.42%	63.59%	63.78%
PDFBox-1.8.4	54.44%	50.43%	49.17%	42.86%
Commercial	119.05%	83.31%	80.67%%	78.27%
Accept	$\alpha = 0.01$	$\alpha = 0.04$	$\alpha = 0.01$	$\alpha = 0.02$
	H_1	H_1	H_1	H_1

Table IV answers the second research question. Since this question concerns individual files, we only consider the 4 file-level patterns in the quantitative analysis, that is: *Unstable Interface*, *Implicit Cross-module Dependency*, *Unhealthy Inheritance Hierarchy*, and *Cross-Module Cycle*. The remaining pattern, *Cross-Package Cycle* is a package-level pattern.

In Table IV, column $\#AH$ indicates the number of architecture hotspots that a file participates in. The table shows that the more architecture hotspots a file is involved in, the more maintenance effort it has caused. Consider Avro as an example, the files involved in four architecture hotspots exhibit an average bug frequency of 16.5, which is significantly higher than the bug frequency of files that are only involved in three hotspots, where the number is is only 7.9.

To answer this question more rigorously, we conducted *Pearson Correlation Analysis* to test the dependency between the number of issues a file involves ($\#AI$) and the average values of its four measures. The PC row, at the bottom of Table IV shows the *Pearson Coefficient* value for each measure. The PC values indicate strong correlations between $\#AI$ and the $avg\_measure_i$. That is, the more architecture issues a file is involved in, the more maintenance effort it will cause.

To answer RQ3 we needed to look at each hotspot pattern independent of the others. Table IV shows the *total* number of hotspots that files are involved in. But we want to analyze

TABLE II: Identified Architecture Hotspots

	Avro-1.7.6		Camel-2.11.1		Cassandra-1.0.7		CXF-2.7.10		PDFBox-1.8.4	
Type	#Instances	#Files	#Instances	#Files	#Instances	#Files	#Instances	#Files	#Instances	#Files
Unstable Interface	2	39	3	2653	3	219	3	104	1	57
Implicit Cross-module Dependency	7	33	77	447	49	249	136	1010	23	137
Cross-Module Cycle	40	34	533	139	11877	192	8054	244	577	152
Unhealthy Inheritance Hierarchy	61	88	303	409	138	190	635	705	239	221
Total	110	124	916	744	12067	448	8828	1504	840	337
	Hadoop-2.2.0		HBase-0.94.16		Ivy-2.3.0		OpenJPA-2.2.2		Commercial	
	#Instances	#Files	#Instances	#Files	#Instances	#Files	#Instances	#Files	#Instances	#Files
Unstable Interface	2	133	3	212	1	85	1	70	2	79
Implicit Cross-module Dependency	35	221	74	370	16	120	55	316	9	136
Cross-Module Cycle	1275	103	4939	203	15095	132	12119	345	25	32
Unhealthy Inheritance Hierarchy	140	236	196	278	104	139	826	719	102	141
Total	1452	469	5212	640	15216	294	13001	958	138	270

TABLE IV: Average values of measures for the subjects

Avro-1.7.6					Camel-2.11.1					Cassandra-1.0.7				
#AH	BF_avg	BC_avg	CF_avg	CC_avg	#AI	BF_avg	BC_avg	CF_avg	CC_avg	#AI	BF_avg	BC_avg	CF_avg	CC_avg
0	0.1	3.7	0.5	29.0	0	0.5	7.9	2.2	58.2	0	0.4	7.1	1.0	32.6
1	0.4	3.9	0.9	26.2	1	1.2	18.5	5.6	131.5	1	1.1	17.4	4.8	106.4
2	1.6	12.6	5.2	376.7	2	3.7	56.6	14.4	304.7	2	5.3	84.5	21.2	559.1
3	7.9	124.5	21.6	628.5	3	8.4	141.5	33.9	681.3	3	12.8	245.8	45.7	1202.0
4	16.5	255.0	33.5	1220.0	4	13.9	204.7	50.9	1043.5	4	18.8	364.9	65.7	1909.4
PC	0.91	0.89	0.94	0.95	PC	0.96	0.96	0.97	0.97	PC	0.97	0.96	0.98	0.97
CXF-2.7.10					Hadoop-2.2.0					HBase-0.94.16				
#AI	BF_avg	BC_avg	CF_avg	CC_avg	#AI	BF_avg	BC_avg	CF_avg	CC_avg	#AI	BF_avg	BC_avg	CF_avg	CC_avg
0	0.8	21.0	2.8	86.9	0	0.4	12.7	1.0	56.8	0	0.7	10.4	0.9	53.0
1	2.9	62.3	9.4	262.5	1	1.5	24.8	4.2	167.7	1	4.8	236.7	8.3	614.6
2	8.6	164.8	23.1	592.0	2	5.3	173.6	13.8	558.3	2	9.9	418.5	17.2	2083.6
3	20.2	390.9	52.5	1232.4	3	26.0	725.1	58.0	1959.6	3	47.8	1335.1	87.6	3158.7
4	54.1	890.2	142.3	3326.0	4	13.7	237.9	26.8	1252.0	4	76.7	2370.4	135.1	6019.0
PC	0.90	0.92	0.89	0.89	PC	0.76	0.63	0.72	0.83	PC	0.93	0.94	0.93	0.97
Ivy-2.3.0					OpenJPA-2.2.2					Pdfbox-1.8.4				
#AI	BF_avg	BC_avg	CF_avg	CC_avg	#AI	BF_avg	BC_avg	CF_avg	CC_avg	#AI	BF_avg	BC_avg	CF_avg	CC_avg
0	0.2	4.5	1.1	31.8	0	1.8	10.0	1.1	36.8	0	0.5	27.1	1.1	92.0
1	1.1	22.8	3.3	79.6	1	3.2	31.1	3.7	111.5	1	1.4	35.9	2.9	136.5
2	2.9	54.6	8.4	251.9	2	4.6	64.5	7.5	229.8	2	1.5	64.1	3.4	259.9
3	7.0	119.9	20.9	646.2	3	10.8	408.6	22.4	862.5	3	8.1	495.0	13.7	861.3
4	6.4	204.6	18.6	792.3	4	25.1	981.0	52.5	2301.1	4	12.2	669.5	18.4	1254.4
PC	0.94	0.96	0.93	0.97	PC	0.90	0.88	0.90	0.88	PC	0.92	0.92	0.94	0.94
Commercial Project														
#AI	BF_avg	BC_avg	CF_avg	CC_avg										
0	0.1	2.25	2.7	102.5										
1	0.2	4.6	5.9	200.4										
2	0.8	3.24	10.3	372.0										
3	2.8	36.8	19.8	884.7										
4	6.0	21	29.0	549.0										
PC	0.91	0.73	0.98	0.81										

which hotspots have the greatest influence on a file's overall error-proneness and change-proneness, and hence which hotspots are the greatest contributors to technical debt. From our analysis, we found that the files that suffered from *Unstable Interface* and *Cross-Module Cycle* are extremely error-prone or change prone.

Tables V-VIII report the increase in the value of the measure attributable to each type of architecture hotspot. The majority of the measures increased, over all projects' all architecture hotspots, as expected. These tables show that the greatest impact in terms of average increase of the measures is attributable to *Unstable Interface* and *Cross-Module Cycle*. That is to say that, while all of the hotspots contribute to bug frequency, change frequency, bug churn and change churn, *Unstable Interface* and *Cross-Module Cycle* contribute the most by far.

In summary, from our quantitative analysis we can make the following observations: first, the hotspot instances we detect capture file groups with significantly higher error-proneness and change-proneness than average project files. Second, when a file is involved in higher numbers of architecture issues, its error-proneness and change-proneness increase significantly. Third, the files suffering from the *Unstable Interface* hotspot type are more error-prone and change-prone than

TABLE V: Increase in the measures' average values for the files involved in Unstable Interface

Subjects	BF_inc	BC_inc	CF_inc	CC_inc
Avro-1.7.6	481.76%	465.36%	451.65%	401.94%
Camel-2.11.1	308.54%	299.26%	304.08%	268.38%
Cassandra-1.0.7	312.53%	312.29%	310.03%	307.35%
CXF-2.7.10	641.14%	527.52%	562.38%	466.19%
Hadoop-2.2.0	767.42%	865.54%	731.28%	650.65%
HBase-0.94.16	941.09%	998.32%	1005.96%	912.11%
Ivy-2.3.0	262.14%	269.47%	256.18%	282.43%
OpenJPA-2.2.2	421.47%	1159.18%	652.94%	825.40%
PDFBox-1.8.4	342.27%	432.11%	280.62%	285.49%
Commercial	555.54%	367.57%	269.49%	248.42%

TABLE VI: Increase in the measures' average values for the files involved in Cross-Module Cycle

Subjects	BF_inc	BC_inc	CF_inc	CC_inc
Avro-1.7.6	309.68%	310.99%	265.64%	205.70%
Camel-2.11.1	241.43%	247.97%	204.94%	196.65%
Cassandra-1.0.7	165.94%	180.84%	159.03%	168.17%
CXF-2.7.10	275.71%	253.40%	231.89%	194.62%
Hadoop-2.2.0	432.89%	426.76%	405.52%	335.27%
HBase-0.94.16	489.95%	412.98%	511.54%	313.78%
Ivy-2.3.0	190.31%	236.15%	143.37%	161.60%
OpenJPA-2.2.2	75.37%	231.42%	147.87%	175.15%
PDFBox-1.8.4	104.83%	142.75%	82.43%	87.30%
Commercial	117.44%	-8.32%	32.59%	80.09%

TABLE VII: Increase in the measures' average values for the files involved in Unhealthy Inheritance Hierarchy

Subjects	BF_inc	BC_inc	CF_inc	CC_inc
Avro-1.7.6	75.51%	77.00%	68.76%	53.25%
Camel-2.11.1	65.64%	67.00%	62.63%	59.81%
Cassandra-1.0.7	115.23%	116.20%	103.99%	112.42%
CXF-2.7.10	100.38%	96.57%	82.68%	75.90%
Hadoop-2.2.0	140.47%	113.26%	111.73%	82.86%
HBase-0.94.16	345.44%	285.83%	356.43%	212.19%
Ivy-2.3.0	108.46%	108.16%	92.27%	101.67%
OpenJPA-2.2.2	40.62%	84.67%	63.77%	64.00%
PDFBox-1.8.4	82.67%	99.87%	65.34%	75.67%
Commercial	106.36%	143.90%	53.23%	91.39%

the files suffering from the other architecture issues. However, almost all of the measures in Tables V-VIII increased, which means that all of the hotspots contribute to error-proneness and change-proneness.

D. Qualitative Analysis

We conducted a qualitative analysis to evaluate whether our architecture hotspot detection approach can capture the architecture problems that an industrial architect considers to be important and worth refactoring. We also wanted to know if we could capture architecture problems that could not be captured by other tools. Most importantly, we wanted to know if the detected hotspots, in the form of DRSpace DSMs, provided useful guidance to the architect about how to refactor.

We conducted a case study with a software company which, for reasons of confidentiality, we refer to as "CompanyS". CompanyS shared project data with us for a project that we refer to as "Commercial" [16]. Commercial has evolved for about 4 years, with approximately 800 files and 56,000 SLOC. The data received from CompanyS included the dependency information reverse engineered by Understand, SVN logs, and issue tracking records for the past two years.

Given these inputs, we used Titan to produce SDSM, HDSM, and DRH clustering files as well as a package clustering file. Then we used our hotspot detector to determine all instances of hotspot patterns. We identified 3 groups of files in Commercial with Unhealthy Inheritance Hierarchy instances (UIH1 with 6 files, UIH2 with 3 files, and UIH3 with 7 files), one group of files with Implicit Cross-module Dependency (ICD with 27 files), and two groups of files exhibiting Unstable Interfaces (UI1 with 26 files, UI2 with 52 files). We communicated these issues, in the form of 6 DRSpaces, back to the chief architect of CompanyS. We then asked the architect a number of questions for each of the issues identified.

TABLE VIII: Increase in the measures' average values for the files involved in Implicit Cross-module Dependency

Subjects	BF_inc	BC_inc	CF_inc	CC_inc
Avro-1.7.6	34.96%	-15.09%	51.70%	-15.12%
Camel-2.11.1	7.44%	6.76%	12.43%	5.88%
Cassandra-1.0.7	11.09%	7.73%	15.27%	1.43%
CXF-2.7.10	11.29%	11.55%	16.43%	9.70%
Hadoop-2.2.0	45.83%	74.43%	41.21%	55.25%
HBase-0.94.16	26.72%	42.14%	24.40%	105.04%
Ivy-2.3.0	-14.93%	-46.46%	7.29%	0.71%
OpenJPA-2.2.2	-4.88%	-14.27%	14.94%	21.73%
PDFBox-1.8.4	36.53%	-6.04%	42.06%	-9.92%
Commercial	165.11%	119.10%	82.54%	74.39%

First we asked whether each problem was, according to him, a real architectural problem? The architect responded that, of all the hotspots that we found, only UIH3 with 7 files has low maintenance costs. All of the other hotspots were confirmed by the architect to be significant, high-maintenance architecture issues.

Next we asked whether they planned to refactor and fix these issues in the near future? The architect indicated that they planned to refactor all five of the "serious" architecture issues, with UIH2 , ICD, and UI1 being given the highest priority. They also planned to refactor UHI and UI2 if they have time.

Finally, we asked whether any of the issues we identified were not revealed by the other tools that CompanyS used? The architect pointed out that the Implicit Cross-module Dependency we identified revealed a fundamental problem that was not detectable by other tools. This revealed a poor design decision that caused a large number of co-changes among files that had no structural relationship.

This feedback from CompanyS is extremely encouraging. Furthermore, each instance hints at a corresponding refactoring strategy. For example, an Unhealthy Inheritance Hierarchy instance is displayed in a DSM with just the files involved in the hierarchy, so their relations can be easily checked. If the problem is that a parent class depends on one of its children, then it is obvious that the classes should be refactored, perhaps by moving some code from the child to the parent class, to remove this improper dependency.

When one of the Unstable Interface instances was reported, our collaborator realized that the interface was poorly designed; it was overly complex, and had become a God interface. Divide-and-conquer would be the proper strategy to redesign the interface. Based on the analysis results we provided, CompanyS is currently prosecuting a detailed refactoring plan to address these issues one by one.

Our qualitative evaluation results are also encouraging: the fact that our collaborators not only confirmed the significance of the architecture issues we reported, but also initiated actions to refactor them one by one, demonstrated the effectiveness of our tool. More importantly, the DSMs that visualize these instances provided direct guidance about how each architecture problem should be addressed.

VI. Discussion

We now turn to a brief discussion of the limitations of our tool, and threats to validity. First, although we have defined five architecture hotspot issues in this paper, not all of them are equally easy to calculate. Two of them: *Unstable Interface* and *Implicit Cross-module Dependency* crucially depend on the availability of the project's evolution history. Thus, for projects where this information is not available, it is impossible to detect these architecture issues. Furthermore, when detecting these two architecture issues, the results are related to the selection of thresholds. Changing the value of the thresholds will change the result sets. Determining the best thresholds and sensitivity of the results to threshold values is ongoing work.

Second, while our tool is scalable in terms of the number of hotspots that it can detect, at the moment we have just five. We have no way of estimating how much of the space of possible architectural issues these five hotspots cover. The good news, however, is that when we do identify more types of architecture issues (revealing structural problems) it is easy to add these new types into our detection tool. Exploring more types of architecture issues is therefore part of our future work.

Third, a threat to validity is in our data set. Because we only selected nine Apache open source projects and one commercial project to analyze, we can not yet say with complete confidence that our results are generalizable across all software projects. However, we chose projects of different sizes and domains to partially address this issue. Furthermore, we applied our tool on both open source and commercial projects. We are in the process of applying our tool to more projects, to further bolster the robustness of our results.

Finally, we can not guarantee that the four bug and churn measures that we chose are the best proxies for maintenance effort. We are, however, currently working with a commercial project that is providing true effort data. We intend to report on the results of this analysis in the future, to further demonstrate the effectiveness of our tool and to validate the use of the proxy measures.

VII. Related work

In this section we compare our approach with the following research areas.

Defect Prediction and Localization: Selby and Basili [25] have investigated the relation between dependency structure and software defects. There have been numerous studies of the relationship between evolutionary coupling and error-proneness [8], [11], [5]. For example, Cataldo et al.'s [5] work reported a strong correlation between density of change coupling and failure proneness. And Ostrand et al. [23]'s study demonstrated file size and file change information were very useful to predict defects. The relation between various metrics and software defects has also been widely studied. For example, Nagappan et al. [22] investigated the different complexity metrics and demonstrated that these metrics are useful and successful for defect prediction. However, they also reported that, in different projects, the best metrics for prediction may be different. Besides, defect localization has also been widely studied [14], [12], [19]. For example, Jones et al. [14] used the ranking information of each statement to assist fault location.

However, all the above works focus on individual files as the unit of analysis, but do not take architectural structure into consideration. Our study focuses on the architectural issues which reveal problems in the software. Both file complexity and architectural complexity are, we conjecture, contributors to overall error-proneness in a software system.

Code Smell Detection: Fowler [7] describes the concept of a "bad smell" as a heuristic for identifying refactoring opportunities. Code clones and feature envy are examples of well-known bad smells. Similarly, Garcia et al.'s [10] study reports some architectural bad smells. Automatic detection of bad smells has been widely studied. For example, Moha et al. [20] presented the Decor tool and language to automate the construction of design defect detection algorithms. There have been a number of proposals for automatically detecting bad smells which may lead to refactorings. For example, Tsantalis and Chatzigeorgiou's study [28] presented a static slicing approach to detect extract method refactoring opportunities. For some specific bad smells, such as code clones, there has been substantial research on their detection. For example, Higo et al. [13] proposed the Aries tool to identify possible code clones leading to potential refactorings.

Our hotspot detection approach is different. First, our approach focuses on recurring architecture issues, instead of being confined to types of bad smells. Furthermore, multiple bad smells are, we claim, instances of architecture issues. Second, bad smells have been detected in a single version of the software, while our approach considers a project's evolution history. In this way we can focus on the most recent and frequently occurring architecture problems, and we can detect problems that span multiple releases, such as Implicit Cross-module Dependency and Unstable Interfaces, neither of which could be detected by looking at a single snapshot of a code base.

Architecture Research: Our work is also related to research on software architecture representation and analysis. There has been substantial study on the uses of architecture representations (views) [2], [18], [27], [6] and how they support design and analysis. For example, Kruchten [18] proposed the 4+1 view model of architecture. Our architecture representation, DRSpace, focuses on just a single architecture view—the module view. Within the module view DRSpaces are organized based on design rules and independent modules.

The analysis of architecture has also been widely studied. Kazman et al. [15] created the Architecture Tradeoff Analysis Method for analyzing architectures. Andrew [17] proposed the anti-pattern to represent recurring problems that are harmful to software systems. These methods depend, however, on the skill of the architecture analysts. Our approach, by contrast, can detect the architecture issues automatically and guide the user, helping them to diagnose software quality problems.

VIII. Conclusion

In this paper, we have formally defined five architecture issues—called *hotspot* patterns—that appear to occur ubiquitously in complex software systems. The identified hotspots pinpoint structural problems that contribute to technical debt—bugginess, change-proneness, and increased costs of fixing

bugs and making changes. We proposed a formalism to represent these hotspots, and a novel tool to automatically detect and identify the files involved in these hotspot patterns. We evaluated our tool by examining nine Apache open source projects and one commercial project. Our results show that hotspot patterns are strongly correlated with higher bug frequencies, change frequencies, and with the effort to fix bugs and make changes. We also provided evidence that the more architectural issues a file is involved with, the more likely it is error-prone, and the more effort it takes to fix and modify. And we found that the *Unstable Interface* and *Cross-Module Cycle* hotspot patterns contributed the most to a file's error-proneness and change-proneness. Finally, we presented a qualitative analysis wherein we demonstrated that our approach is effective in helping real-world developers find the important structural problems that contribute to high maintenance effort. More than just finding the problems, the identified hotspots guide them in conducting their refactoring. This research therefore provides not just crucial insight into reducing the costs and risks of long-term software sustainment, but provides a tool that helps in automating the analysis.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation of the US under grants CCF-0916891, CCF-1065189, CCF-1116980 and DUE-0837665.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This material has been approved for public release and unlimited distribution. DM-0002087

REFERENCES

[1] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press, 2000.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 3rd edition, 2012.

[3] Y. Cai and K. J. Sullivan. Modularity analysis of logical design models. In *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 91–102, Sept. 2006.

[4] Y. Cai, H. Wong, S. Wong, and L. Wang. Leveraging design rules to improve software architecture recovery. In *Proc. 9th International ACM Sigsoft Conference on the Quality of Software Architectures*, pages 133–142, June 2013.

[5] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6):864–878, July 2009.

[6] D. Falessi, G. Cantone, R. Kazman, and P. Kruchten. Decision-making techniques for software architecture design: A comparative survey. *ACM Computing Surveys*, 43(4):1–28, Oct. 2011.

[7] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, July 1999.

[8] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. 14th IEEE International Conference on Software Maintenance*, pages 190–197, Nov. 1998.

[9] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[10] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *Proc. 13th European Conference on Software Maintenance and Reengineering*, pages 255–258, Mar. 2009.

[11] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[12] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 263–272, 2005.

[13] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring support based on code clone analysis. In *Proc. 5th International Conference on Product Focused Software Development and Process Improvement*, pages 220–233, Apr. 2004.

[14] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. 24thInternational Conference on Software Engineering*, 2002.

[15] R. Kazman, M. Barbacci, M. Klein, S. J. Carriere, and S. G. Woods. Experience with performing architecture tradeoff analysis. In *Proc. 16th International Conference on Software Engineering*, pages 54–64, May 1999.

[16] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka. A case study in locating the architectural roots of technical debt. In *Proc. 37th International Conference on Software Engineering*, May 2015.

[17] A. Koenig. *Patterns and antipatterns*. The patterns handbooks, 1998.

[18] P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12:42–50, 1995.

[19] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *13rd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2005.

[20] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, and L. Duchien. A domain analysis to specify design defects and generate detection algorithms. In *Proc. 11th International Conference on Fundamental Approaches to Software Engineering*, pages 276–291, Mar. 2008.

[21] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proc. 30th International Conference on Software Engineering*, 2008.

[22] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. 28th International Conference on Software Engineering*, pages 452–461, 2006.

[23] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[24] R. Schwanke, L. Xiao, and Y. Cai. Measuring architecture quality by structure plus history analysis. In *Proc. 35rd International Conference on Software Engineering*, pages 891–900, May 2013.

[25] R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, Feb. 1991.

[26] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *Proc. Joint 8th European Conference on Software Engineering and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 99–108, Sept. 2001.

[27] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[28] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, May 2009.

[29] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In *Proc. 33rd International Conference on Software Engineering*, pages 411–420, May 2011.

[30] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi. Design rule hierarchies and parallelism in software development tasks. In *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 197–208, Nov. 2009.

[31] L. Xiao, Y. Cai, and R. Kazman. Design rule spaces: A new form of architecture insight. In *Proc. 36rd International Conference on Software Engineering*, 2014.

[32] L. Xiao, Y. Cai, and R. Kazman. Titan: A toolset that connects software architecture with quality analysis. In *22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2014.